Blocking Java Applets at the Firewall

David M. Martin Jr.[1]
Department of Computer Science
Boston University
111 Cummington Street
Boston, MA 02215
dm@cs.bu.edu

Sivaramakrishnan Rajagopalan
Bellcore
445 South St.
Morristown, NJ 07960
sraj@bellcore.com

Aviel D. Rubin
Bellcore
445 South St.
Morristown, NJ 07960
rubin@bellcore.com

Abstract

This paper explores the problem of protecting a site on the Internet against hostile external Java applets while allowing trusted internal applets to run. With careful implementation, a site can be made resistant to current Java security weaknesses as well as those yet to be discovered. In addition, we describe a new attack on certain sophisticated firewalls that is most effectively realized as a Java applet.

1. Introduction

Java is hot stuff. The hype that has surrounded Java and Java-enabled browsers such as Netscape mirrors the hype associated with the Internet and the World Wide Web. It is inevitable that this trend will not only continue, but that the acceptance and use of Java will grow. In spite of the many security concerns about using Java within browsers [8, 11], the momentum that has resulted from the convenience and functionality of downloadable executables has taken the Internet by storm.

Flaws in the design and implementation of Java-enabled browsers have repeatedly been discovered. Some of these exploit weaknesses in the type checking of Java, while others exploit system-level bugs. These vulnerabilities can allow Java applets to erase files, leak sensitive information, and corrupt a user's environment. At the very least, malicious applets can cause great inconvenience.

Dimly aware of the dangers, most users are nonetheless unwilling to disable Java in their browsers, and an "It won't happen to me" attitude prevails. One contribution of this paper is the description of an attack that allows an invading Java applet to convince certain firewalls to open arbitrary TCP holes to the applet's host. A variant of this attack even allows the attacker to use a firewall's proxying software—intended to protected the site from intruders—as a springboard for further attacks on the internal network. Either way, by simply visiting a Web page or reading email in a Java-enabled browser, users can unknowingly provide their attacker a route through the protecting firewall.

While Java presents security concerns to users who download applets from the Internet, it is nonetheless very useful as a programming language. Java applets are being developed and deployed for internal use in corporate and other networks at a rapid pace [14]. There are many reasons why people want to run Java within their protected network. Java is platform independent, so multi-platform demos can be built with relative ease. The object-oriented features of Java and the inheritance mechanism it provides make it an easy and convenient programming language, and the many available class libraries make things like network programming much easier than before.

A site wishing to protect itself from hostile Java applets has few options. If users are required to disable Java in their Web browsers, then that site is missing out on all of the benefits of a new and useful programming language, or

[1] This work was performed while the author was a summer student at Bellcore.

at least those features that make it convenient to run within a browser. Also, there is little to stop a user from independently obtaining a Java-enabled browser from the Internet and running it. Cutting the site off from the Internet entirely would probably not sit well with most users.

In this paper, we discuss a compromise. We describe various techniques to block Java applets at a site's firewall so that internal applications can run Java in their browsers, but untrusted applets from the outside cannot penetrate. These techniques are already partially available in the public domain [7], and commercial firewall developers are rapidly releasing products that offer the techniques in various combinations.

Section 2 contains a review of basic firewall concepts. Section 3 describes an applet attack on certain firewalls. In Section 4, an applet's path from server to host is described, and Section 5 evaluates three common methods for blocking Java applets at the firewall. Section 6 briefly describes the authors' experimental implementation of a Java-blocking firewall.

2. Firewall Design

A firewall is an intentional bottleneck between two networks designed to prohibit certain types of internetwork communication such as login attempts and network file system access [6, 5, 1, 3, 10, 15, 16, 20]. The firewall hardware typically consists of one or more computers, routers, or special-purpose machines. In this paper, we envision a firewall protecting a corporate network as a single dual-homed host, i.e., a computer with two network interfaces able to examine traffic attempting to cross from one interface to another in order to decide whether to permit the traffic flow. The particulars of the firewall's network connectivity, such as screened host versus screened subnet architecture [5], precise location of the decision-making host, etc., are not crucial to the line of reasoning we wish to pursue, so we leave these details unspecified. We assume that one of the firewall's two network interfaces connects to the trusted but vulnerable local network, and the other connects to the external and untrusted Internet. Since one of the networks is the Internet, we restrict our attention to data in the form of IP packets.

Computers *behind* the firewall are the *local hosts* that the firewall protects, and computers *outside* the firewall are the *remote hosts*, which we assume to be potential attackers. TCP connections across the firewall that originate from the Internet are called *inbound* connections, and those that originate behind the firewall are called *outbound* connections; in each case, TCP permits full-duplex communications.

When designing a firewall, the essential decision to be made is what level of granularity to use when regulating the traffic flow. The granularity choice has a significant impact on equipment costs, operating costs, throughput, and security. Fine-grained, lightweight designs operate at the IP packet level and are called *packet filters*. Coarser-grained, heavyweight designs usually operate at the TCP/UDP level and are called gateways or *application proxies*. We discuss both approaches in the following sections.

2.1. Packet Filters

A packet filtering firewall examines each IP packet independently, and by examining the IP source, IP destination, and other fields, decides whether to forward the packet or not. Ordinary packet filters make this decision by matching the packet against a static set of simple rules. For instance, one rule could prevent NFS packets from crossing the firewall in either direction, and another rule could block and log the initial packets of inbound TCP telnet port connections, where “initial” means that `ACK=0` in the packet's TCP options field. The latter rule allows local hosts to telnet to Internet hosts, but prevents Internet hosts from telnetting into the local hosts. By blocking only the initial packets of a TCP session, such a packet filter implicitly relies on the local hosts to safely discard remaining packets when the initial packet doesn't arrive. This in turn relies on the assumption that all local hosts are trustworthy.

More sophisticated packet filters allow the administrator to specify complex rules based on previous packet contents, time elapsed, and other parameters; these are sometimes called *dynamic* packet filters.

Firewalls consisting primarily of packet filters are used at many sites. However, packet filters cannot easily enforce policies such as “only users in group X may telnet from the Internet to local hosts.” The problem is that a user's membership in group X is a property that has to be described in a relatively high-level protocol. Because IP packets may be arbitrarily fragmented and reordered and packet filters do not retain (much) state between packets, another solution is required if the policy is to be enforced at the firewall. Indeed, many useful protocols such as FTP, X11, and talk are difficult although not impossible to provide in a secure fashion using a packet filter alone.

2.2. Application Proxies

Application proxies can be thought of as man-in-the-middle session forwarders. In this firewall scheme, a router or simple packet filter is configured to forward all relevant packets to a secured *proxy host*. For each protocol to be supported across the firewall, a user-mode program listens at the appropriate TCP port[1] on the proxy host. Such programs—the individual *proxies*—implicitly use the proxy host's kernel to reassemble IP packets into TCP streams.

[1] Or UDP port. For our purposes, we concentrate on the TCP case.

Not only are these streams considerably easier to parse and forward than raw IP, but unwelcome packets will rarely make it deeper into the local network than the proxy host. The effect is to isolate the local network from the Internet in a very strong way.

For example, the “only group X may telnet” policy above is easily implemented using an application proxy: the proxy immediately presents an authentication challenge to users attempting inbound telnet from the Internet. Once authenticated, the proxy opens a telnet connection to the local host and binds the two connections. The remote user seems to have a direct connection to the desired equipment, but all of the data actually flows through the proxy host. In this mode, the proxy acts as a *tunnel* through the firewall.

3. A New Attack

We have discovered a new attack against certain firewalls that is most appropriately realized as a Java applet. The success of the attack depends on the firewall's ability to parse and react to an FTP control session in a particular way so that local hosts can use FTP to obtain files from the Internet. In addition, we assume that the firewall is *transparent*, i.e., it intercepts and forwards the TCP streams opened by the browser without the browser's knowledge or cooperation. Both transparent packet-filtering and proxying firewalls are at risk, at least in principle. Ironically, transparent firewalls tend to be the most sophisticated ones.

We strongly recommend that firewall administrators request updates from their vendors. While this attack by itself may or may not concern firewall administrators, it is certainly a good indication of the security risks of allowing local hosts to execute a program chosen by an adversary, even under the severe functionality restrictions imposed upon Java applets.

3.1. FTP and `PORT`

To explain the attack, we first review how FTP ordinarily works [17]. A local user connects to a remote FTP server (port 21) and is authenticated, say as the “anonymous” user. To retrieve the file `/file.txt`, the user's FTP client first chooses an arbitrary TCP port where it will wait for the file to arrive. The client then sends a `PORT` command to the server announcing this choice, and only then issues the command to fetch the file. The server responds by actively opening an inbound TCP connection to the specified port on the client and transmitting the file on this connection. See Figure 1 for a transcript of a sample session.

If the client's network is protected by a simple packet filter, the standard FTP transfer scenario will fail when the server attempts to open the agreed-upon port of the client. To a packet filter, this seems like an unsolicited probe of the client. However, a firewall susceptible to our attack contains special code that watches for precisely such a `PORT` command. In response, the firewall creates a window of opportunity for the remote host to open the named port on the client. If the remote host establishes this inbound connection, it can stay open an arbitrarily long time in order to transfer the file.

```
220 mrr96 FTP server (SunOS 4.1) ready.
USER anonymous
331 Guest login ok, send ident as password.
PASS ftp
230 Guest login ok, access restrictions apply.
PORT 172,16,1,1,19,233
200 PORT command successful.
RETR /file.txt
150 ASCII data connection for /file.txt
    (172,16,1,1,5097) (107 bytes).
226 ASCII Transfer complete.
QUIT
221 Goodbye.
```

Figure 1. A sample FTP session. Lines beginning with numbers are server responses. The contents of `/file.txt` arrive on port $5097 = 19 * 256 + 233$, not on the control port shown here.

3.2. Taking Advantage

But the firewall has no way of knowing that it was a legitimate FTP client that generated the `PORT` command. Suppose that a malicious insider opens an FTP connection to a remote accomplice and announces that it intends to receive a transmission on port 23—the telnet port. The firewall will dutifully open a hole allowing the remote host to access the client's telnet port. The client, in response to the inbound telnet connection, will issue a login prompt as always, giving the accomplice an opportunity to attempt a login in spite of the firewall. We have verified that this attack works in some environments. Of course, it does amount to an insider attack.

3.3. A Java Realization of the Attack

However, the attack can also be realized as a Java applet. Java provides an almost imperceptible means to get attacking code running on the client—a user need only stumble onto the wrong Web page or read email or news in a Java-enabled browser—and from that point, the firewall treats that code as a trusted entity. That is, *permitting the unre-*

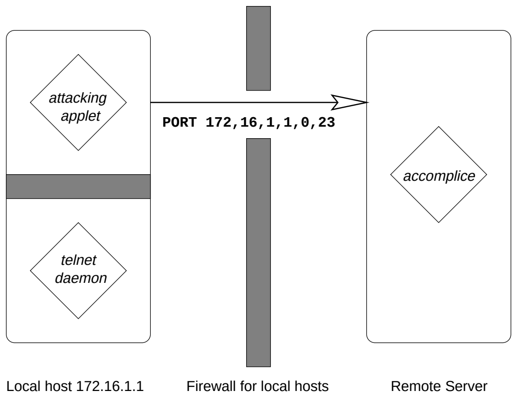

Figure 2. First step. Java's `SecurityManager` **prevents the attacking applet from accessing the local telnet port directly. The applet instead sends the** `PORT` **command to an FTP control port on the accomplice server. The firewall permits the outbound connection.**

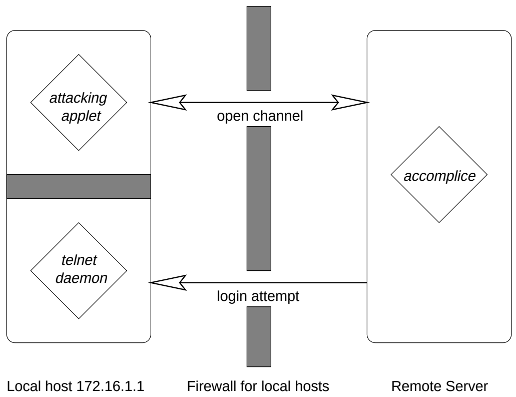

Figure 3. Second step. Having noticed the `PORT` **command, the firewall gives the remote host an opportunity to open the named port—the telnet port (23)— supposedly for a file transfer. The local host reacts to the open by issuing the accomplice a telnet login prompt. The firewall was supposed to prevent this.**

stricted transfer of Java applets totally automates the insider's role in this attack. The user controlling the browser need never even know that the attack took place.

When a victim obtains and invokes our applet from an attacking machine, the applet issues an appropriate `PORT` command and instructs a server running on the attacking machine to attempt a telnet session to the victim's machine. See Figures 2 and 3. The applet is written in perfectly legal Java, and the server is a simple Perl script. *Our applet can bypass susceptible firewalls to penetrate any TCP port on the client.*

3.4. Discussion of the Attack

This attack has been independently discovered by other researchers [2, 8] and was alluded to in section 5.1 of [8]. Indeed, the fundamental weakness being exploited in the attack—FTP's use of inbound TCP connections for receiving files—has long been recognized as a special problem for firewalls [6, 5, 3].

Before the advent of portable executable formats, every process running inside the firewall was seen to be authenticated (albeit weakly) as an insider, so permitting unrestricted outbound FTP didn' t conflict with the security policy stating "that which is not explicitly permitted is forbidden." The `PORT` command crossing the firewall could be taken as proof that the insider initiated the transfer, and so the inbound TCP connection delivering the file was assumed to be associated with the permitted outbound FTP session. However, now that Java-enabled Web browsers invite adversaries to determine the insider's actions, such an assumption is no longer valid. If users can run applets obtained from outside the firewall, then the firewall must treat unauthenticated insiders as adversaries. This changes the firewall's role in the security landscape dramatically. It is particularly interesting to note that even under the assumption that the Java and firewall systems are each independently secure, they do not compose to produce a secure system.

3.5. Partial Solutions

Sites that require strong authentication (e.g., a challenge-response sequence) of users attempting outbound FTP are not susceptible to this attack, since an applet will not be able to authenticate itself as the user. While strong authentication is often required for *inbound* FTP connections, some designers reason that since inside users are already authenticated, no separate authentication should be required for outbound connections. See [1] for an example of a high-security environment in which this reasoning was used. (Note that this paper predates Java-enabled browsers.)

Currently susceptible firewalls can be strengthened by modifying them to ignore `PORT` commands that announce transfers to any "well-known" ports [18] (including ports less than 1024, the so-called "privileged ports"), at the risk of colliding with FTP sessions that innocently happen to choose one of these ports for a file transfer. If an innocent collision occurs, a subsequent attempt will probably succeed; only a tiny fraction of the port space is well-known.

Of course the firewall must be also made aware of any sensitive local applications listening on non-well-known ports.

Another method of decreasing vulnerability is to allow only *passive mode* FTP across the packet filter [17, 3]. The passive mode transfer scenario works almost like the standard mode, except that the *server* chooses a port number to listen on, and the *client* actively opens that port. (TCP connections are always full-duplex, so it makes little difference which end opens the connection.) In this way, the firewall sees the client opening an outbound connection and permits it; no fancy `PORT` command interpretation is required. In particular, our Java applet then sends its malicious `PORT` command in vain, for the firewall ignores it. Unfortunately, some FTP servers do not properly implement passive mode, and so this is not always an option. In this case, the network administrator could establish an FTP proxy that understands passive mode, use a packet filter to forbid all FTP packets across the firewall unless one of the hosts is the proxy host, and disable any special reaction to `PORT` commands. The proxy would then be able to access FTP servers that do not understand passive mode on the clients' behalf. If the FTP proxy is not transparent, then the FTP clients on the local hosts will have to be modified (or the users retrained to use the existing clients in a different way) so that communication with the proxy is possible.

3.6. Variants of the Attack

Bellovin [2] points out that the attack may even be possible without the use of Java. If a browser can be coaxed into sending an HTTP request containing a properly formatted `PORT` command to a server's FTP port, for instance by uploading a multipart form to `http://evil.com:21/`, then the firewall will open the corresponding TCP hole. Netscape will not honor a URL indiciating an “unnatural” low-numbered port (such as HTTP over port 21), so Netscape cannot be used to carry out the attack. However, this approach should work with other Web browsers.

Our attack relies on the assumption that the firewall is “transparent”, i.e., it intercepts and forwards the TCP streams opened by the browser without the browser's knowledge or cooperation. At other firewalled sites, users are instead required to explicitly configure their browsers to use a forwarding proxy, and packet filters forbid communications across the firewall except for those addressed to or from the proxy host. The browser can therefore only reach the outside network by connecting to the proxy. Suppose that the firewall at site `xxx.com` is designed in this manner and has its proxy listening on `proxy.xxx.com` at port 1200. When instructed to fetch a URL of the form `http://evil.com/path`, the browser instead opens a TCP connection to `proxy.xxx.com:1200` and issues the request `GET http://evil.com/path HTTP/1.0`. (The string `HTTP/1.0` identifies the protocol in use.) The proxy then forwards the request to `evil.com` and returns the result to the browser.

If a browser uses a proxy to fetch an applet from `http://evil.com/PokeHole.class`, it then informs the Java `SecurityManager` that the applet “came from” `evil.com`, even though the browser actually obtained it directly from `proxy.xxx.com`. That is, the browser trusts that the proxy did what it was supposed to do. See [4] for details on proxying in HTTP/1.0.

Our attack fails when explicit proxying is used, because the `SecurityManager` prevents the applet from opening a TCP connection to any host other than `evil.com`, but the site's packet filter prevents the browser from directly connecting to that host. Therefore, the FTP connection for the malicious `PORT` command can never be constructed. (There is no `SecurityManager` equivalent restricting TCP connections in Bellovin's suggested attack above, so it will probably still succeed under explicit proxying.)

Another pernicious attack is possible in an explicit proxying environment. Our assumptions are that an attacker knows or can find out the HTTP proxy host and port, say `proxy.xxx.com:1200` as above, that the victim's browser is configured to use this proxy, and that the proxy's configuration does not prohibit inbound HTTP requests from the proxy itself. Suppose the browser is instructed to load an applet from the bizarre URL `http://proxy.xxx.com:1200/http://evil.com/PokeHole.class`. Then:

1. The browser opens a TCP connection to its configured proxy host `proxy.xxx.com:1200` and issues the request `GET http://proxy.xxx.com:1200/http://evil.com/PokeHole.class HTTP/1.0`.

2. The proxy, in response to the TCP open and `GET` command from step 1, opens a TCP connection to `proxy.xxx.com:1200` (the host and port specified in the `GET` command) and requests the resource named `http://evil.com/PokeHole.class`. That is, it writes `GET http://evil.com/PokeHole.class HTTP/1.0` onto the new TCP connection.

3. The proxy, in response to the TCP open and `GET` from step 2, opens a TCP connection to `evil.com` and requests the resource named `PokeHole.class` by writing `GET /PokeHole.class HTTP/1.0`.

4. The host `evil.com` complies and returns the applet class file, which then ripples back through the two chained proxy sessions to the browser.

5. The browser, examining the original URL requested, informs the Java `SecurityManager` that the applet

"came from" the host proxy.xxx.com, and starts the applet.

Therefore the applet actually supplied by evil.com seems to have come from proxy.xxx.com, and so it is (only) able to open TCP connections to the host proxy.xxx.com. But the proxy host is *designed* to forward TCP streams, so the applet can repeat the proxy self-reference trick to access arbitrary hosts supposedly protected by the firewall. We have verified that this attack works in some proxying environments: our applet, delivered by evil.com, can bounce off the proxy host to access any host reachable from the proxy.

Although this "arbitrary" access to hosts is limited to the protocols supported by the proxy, the proxy's implementation of these protocols may well be rich enough to support a wide array of probes and attacks. For instance, we have seen one proxy that implements a complete two-way tunnel between client and server when forwarding an HTTP request, even though HTTP version 1.0 doesn't support persistent connections. The tunnel is a nice bit of code modularity, but results in bad security; it's enough to carry a telnet session.

We do rely on the assumption that the proxy allows connections from itself in step 3; without it, the attack fails. Most proxies give the firewall administrator great freedom in determining which hosts may connect to the proxy, so the attack can be prevented simply by making sure that proxy.xxx.com rejects connections from proxy.xxx.com. However, it is easy to imagine an administrator configuring the proxy to allow access from *.xxx.com in the absence of this knowledge, thereby enabling such an attack. (Particularly large sites may have multiple proxies to provide sufficient bandwidth for their users; in such a site, the administrator would have to configure each proxy to deny connections from any other proxy.)

This section has described in detail one attack on susceptible firewalls that is most appropriately realized as a Java applet, and has outlined two variants of the attack also exploiting the firewall-browser trust relationship. These particular flaws can be fixed, but other attacks are probably close behind.

Ultimately, firewalls permitting applet access should treat unauthenticated insiders as adversaries. If that is not possible, then sites should implement a mechanism that reverts their security landscape to the familiar one in which it is an acceptable risk to trust unauthenticated insiders for certain operations. This can be done by blocking applets from crossing the firewall so that only trusted code runs on clients. Then our attack, other existing Java attacks, and *yet-to-be-discovered* Java attacks can be prevented. In order to block applets, we first consider the mechanism that brings an applet to a client.

```
<img src=about:javalogo.gif>
Below is a destructive Java applet.
<applet
codebase="ftp://xxx.com/pub"
code=PokeHole
height=100 width=300> </applet>
```

Figure 4. An enabling document.

4. Delivery of Java Applets

We are concerned with Java applets that are fetched, either intentionally or unintentionally, by Web browsers such as Netscape versions 2.02 and 3.0. For the purposes of this paper, "Netscape" refers to either version. Our remarks generally apply to other Java-enabled browsers as well.

In current browser environments, an applet is loaded and invoked in two distinct stages.

4.1. Enabling Document

An applet is requested with HTML code such as that shown in Figure 4. In order to start the applet, the *enabling document* shown in Figure 4 must first be delivered to the browser. The possible means of delivery include HTTP, FTP, gopher, mail, news, and the client's filesystem. All of these delivery mechanisms have URL encodings, so the user might not know or care how the enabling document is delivered: the user only has to do something innocuous, like absentmindedly click on a link. Once the enabling document arrives, the browser parses its HTML code and begins to obtain the remaining pieces, such as the javalogo.gif image and the PokeHole applet class file.

4.2. The Class File

At this point, the browser uses the appropriate delivery mechanism to fetch the URL derived from the codebase and code properties in the <applet> tag. In the above example, the class PokeHole is fetched as ftp://xxx.com/pub/PokeHole.class. When the binary file PokeHole.class is delivered, execution proceeds in its public init() method. Figure 5 shows a hex dump of the beginning of a Java class file. Again, any document delivery mechanism can be used to obtain a class file; however, most untainted news and mail servers would not be capable of delivering the data in the correct format: every file they deliver begins with ASCII header lines, and these would be rejected by the AppletClassLoader. This still leaves HTTP, FTP, gopher, and the client's filesystem as possibilities.

```
00000000: cafe babe 0003 002d 0099 0800 8e08 0098  .......-........
00000010: 0800 9608 008b 0800 9408 0060 0800 5d08  ...........`..].
00000020: 0082 0700 8107 0057 0700 6f07 0085 0700  .......W..o.....
00000030: ...
```

Figure 5. The beginning of a Java class file.

5. Blocking Strategies

In this section we discuss the relative strengths and weaknesses of three strategies to prevent applets from entering the local network through the firewall. In each case we assume that the attacker's goal is to get a Java applet running on the victim's host, where the applet can then exploit existing Java bugs [8, 11] or firewall weaknesses to mount a more serious attack. We assume that completely disabling Java at the local hosts is not an option; perhaps the local use of Java is required in day-to-day-business, or perhaps users may forget to disable Java when they download this week's spiffy new browser beta in a lower-security environment. Besides, there is no reason to disrupt Java applets delivered by trusted machines.

5.1. Rewriting `<applet>` Tags

This strategy uses a proxy that scans enabling documents for `<applet>` tags and rewrites such tags in a benign form, so that the web browser receiving the enabling document does not actually receive `<applet>`, and therefore never even attempts to fetch the attacker's applet. When rewriting the enabling document, the firewall can insert appropriate HTML text explaining that an applet has been blocked. Claunch [7] has extended Trusted Information Systems' free firewall toolkit [12] with this strategy.

This strategy works well in many situations and does prevent most ordinary applets from reaching a victim host. However, in order to locate `<applet>` tags, the firewall must know when to look for the tags and when not to: it would be unacceptable to rewrite such tags if they innocently occurred in a GIF image, a sound clip, or simply in non-HTML text (such as documentation) containing `<applet>`. There are two parts to the "when to look" question:

1. Detecting the `<applet>` tags in HTML documents, and

2. Determining whether the stream contains an HTML document at all.

Part 1 requires the firewall to parse the HTML file in the same way that the browser does. This is a tricky proposition: there are already several Java-enabled browsers, and there is no reason to think that they all produce the same parse trees on a given document. An attacker would only need to find a way to send `<applet>` in such a way that the firewall doesn't notice it but the browser does. Given the wide variety in browser interpretations of HTML elements, the existence of such asymmetry is not implausible. Even worse, supporting `<applet>` tag blocking for multiple browser types means detecting the kind of browser that is requesting a transfer. But at least one Web browser lies about its identity on its `User-agent:` line in order to receive special treatment from servers [9]!

Part 2 can be difficult as well. HTTP version 1.0 servers identify the type of their transmissions using a `Content-encoding:` header line [4]. However, HTTP 0.9, FTP, and gopher servers simply deliver the file, leaving the browser to decide how to interpret the transmission. When the transfer protocol does not identify the file type, browsers examine the extension of the *requested file name* in order to determine a type. For instance, Netscape interprets an HTTP 0.9 delivery of the URL `http://xxx.com/isoc.html` as an HTML file, while it interprets HTTP 0.9 delivery of `http://xxx.com/isoc.gif` as an image file.[2] Therefore, an attacker can fool the `<applet>`-blocking strategy by coaxing a browser into interpreting a file as HTML while making it look non-HTML to the firewall.

5.2. Unwelcome Deliveries

It may be possible, through copious experimentation and non-disclosure agreements, to determine precisely how each browser decides when to invoke its HTML parser. However, investigation would have to be exhaustive—HTML parsing opportunities arise in mysterious ways. For instance, simple experimentation shows that Netscape *with gopher proxying enabled* treats the URL `gopher://xxx.com/`n`/foo` as an HTML resource when $n \in \{0, 5, ;, T, 8, 2\}$, and in many other cases as well, even though there is no reason to expect it: the gopher types listed correspond to "ordinary file", "DOS binary archive", "unknown", "text-based tn3270 session", "text-based telnet session", and "CSO phone-book server". Thus, if

[2] Incidentally, Netscape treats the `.class` filename extension as a *text* specifier: one cannot start an applet by opening the URL of its `.class` file directly.

Netscape directs gopher traffic through an `<applet>`-blocking proxy, the proxy had better treat almost everything as an HTML file, even though in many cases the file isn't HTML at all. In practice, the most likely solution using only `<applet>`-blocking is to forbid gopher traffic across the firewall. This unfortunately tends towards the degenerate definition of a firewall in which all traffic is forbidden; a better solution is required.

Rewriting `<applet>` tags in HTML content as delivered by HTTP (0.9 and 1.0), FTP, and gopher may be possible, but mail and news are also problematic. One simple attack is to search Usenet for postings from the targeted network and identify a victim who uses a Java-enabled browser to read mail and/or news; this information can be inferred from article headers. Once a victim has been identified, the attacker prepares an applet and enabling document and sends the enabling document to the victim (or posts it to the victim's favorite newsgroup). By encapsulating the enabling document with `Content-type: text/html`, the attacker ensures that the browser will parse it as HTML. When the victim reads it, the applet is fetched and invoked.

In order to secure these protocols against applets, every mail message and every news article crossing the firewall must be tested for HTML content, and if found, parsed for `<applet>` tags. This not only puts tremendous load on the firewall, but it suffers from the same asymmetries mentioned above.

These subtleties suggest that the `<applet>`-rewriting strategy alone is probably insufficient but nonetheless very useful as a first line of defense. Although we are primarily concerned with Java applets, it should be emphasized that Netscape's Javascript and Microsoft's ActiveX—the other popular portable-executable formats—deliver the executable in the enabling document proper. *There is no second line of defense for these formats*; if they are to be blocked, they must be blocked in the enabling document.

5.3. Blocking `CA FE BA BE`

As seen in Figure 5 and required by the Java Virtual Machine Specification [13], all Java class files begin with the 4-byte hex signature `CA, FE, BA, BE`. This immediately suggests an applet-blocking strategy: prevent *all* inbound files beginning with the `CA FE BA BE` signature from crossing the firewall. By proxying HTTP, FTP, and gopher, such transfers can be detected and blocked.

Note that in recent versions of Netscape, the `CA FE BA BE` signature does not necessarily have to appear at the beginning of a transmitted file. See §5.4 for an explanation and workaround.

The `CA FE BA BE`-blocking strategy requires inspecting only a tiny portion of file content, so it is efficient and easy to program. It does not require simulating any browser's HTML parsing behavior, nor does it require guessing the type of files crossing the firewall. Recall that mail and news servers are usually unable to deliver proper class files because they prepend ASCII headers to their payloads. As long as the browser user attaches to local and trusted mail and news servers, we can safely ignore the problem of mail and news delivery of class files; therefore, all of the potential transfer protocols for class files are covered. Finally, this strategy is inherently good: it works by detecting a property that Java applets are *required* to have, rather than by detecting properties that are merely consistent with the behavior of existing browsers.

Since this strategy does not consider the type of the file in transit, there is a false-positive risk: legitimate, non-class files could begin with `CA FE BA BE`. We feel that this is less likely in practice than the sequence `<applet`... occurring in a non-HTML file, and an acceptable risk overall.

Naturally, this strategy cannot block Javascript or ActiveX code. The `CA FE BA BE`-blocker would best be applied in combination with the `<applet>`-blocker; if the `<applet>` tag makes it through the first line of defense, then the `CA FE BA BE`-blocker will catch it at the second. In fact, detecting `CA FE BA BE` after the `<applet>`-blocker failed is an excellent sign that someone is trying very hard to sneak in an applet, a condition conceivably worth interrupting a leisurely shower [19].

5.4. Blocking by Requested Filename

Another commonly-suggested strategy is to reject all browser requests via HTTP, FTP, and gopher for files with names ending in `.class`. This strategy once enjoyed most of the advantages of the `CA FE BA BE`-blocker, even though there was never any requirement in the Java Virtual Machine Specification [13] that class file names have the suffix `.class`. Still, we did not find a way to convince versions of Netscape earlier than 3.0 beta 7 to request a class file name that is not so named.

Things have changed. Netscape versions 3.0 beta 7 and later allow Java class files to be encapsulated in an archive file so that multi-class applets can be obtained in one network transaction. (In other words, Netscape decided not to wait for persistent-connection HTTP standards.) The archive format used is Zip without compression, a popular archiving format in the MS-DOS world. See Figure 6 for an example of this delivery method. In the example, the `files.zip` archive contains the `PokeHole` class in one of its class files. A firewall intercepting the browser request would only see a request for the resource named `files.zip`. It would probably be unwise to block all requests for resources having the `.zip` suffix, since Zip files are so widely used. FTP users in particular would find it extremely inconvenient.

```
<applet
codebase="ftp://xxx.com/pub"
archive="files.zip" code=PokeHole
height=100 width=300> </applet>
```

Figure 6. Using the `archive` form. A single network connection fetches the `files.zip` archive, which can contain multiple class files.

One workable technique for blocking applets is to interpret a request for a file beginning with the Zip file signature (or having a name of the form `*.zip`) as an indication that a Java applet *might* be in transit. To find out for sure, the firewall could unpack the archive as it arrives and look for the CA FE BA BE signature. If it finds it, then it could abort the transfer.

Netscape version 3.0 appears to require that archive files have names with suffix `.zip` and that the class files embedded within have names with suffix `.class`. However, this is again merely an observation; Netscape is as free to change this behavior as it was to introduce archive files in the first place. Until then, firewall administrators and designers can only hope to find out about new and dangerous features before attackers do.

5.5. Firewall Environments

Each of the above schemes is most easily realized as an application proxy. None of these schemes can be easily implemented with a packet filter alone, since each requires interpreting a part of the data stream that must be found by context. For example, blocking CA FE BA BE—the simplest scheme—requires searching IP packets for that four-byte signature. However, those four bytes need not arrive in the same IP packet, and if split up, the individual packets may arrive out of order. One might wonder what kind of well-intentioned server or gateway would fragment a packet across its first four bytes, until realizing that in HTTP 1.0, the signature appears after an unpredictable amount of header information. Furthermore, legitimate HTTP traffic doesn' t always occur on the same port; a packet filter would have to determine on-the-fly which ports on which hosts are being used for HTTP.

In spite of this, an application proxy for Web traffic can be added to an otherwise packet-filtering environment. In an academic or “loose-cannon” network, where outbound connections are almost always allowed and users are able and apt to install new browsers, the network administrator needs a way to force Web traffic through the proxy. One social-engineering solution is to use the packet filter to prevent *outbound* connections to port 80, the official [18] and most commonly used HTTP port, thus rendering most Web sites inaccessible. Users will soon seek assistance, at which point the administrator can configure the appropriate proxies in their browsers. This leads to an interesting situation in which the outbound HTTP, FTP, and gopher *protocols* are not prevented (except on port 80), but they are proxied when accessed by a Web browser—the most important time to defuse dangerous Java applets. Other programs that use these protocols would not be impeded; a user who needed to download a Java applet by command-line FTP would not be prevented from doing so. Only the user poised to inadvertently invoke applets would be affected, which fits our design paradigm well.

5.6. Blocking Strategy Weaknesses

1. One must consider the possibility that an enabling document and class file might already exist in the victim's filesystem, perhaps downloaded with documentation in HTML format. Or, if the victim's equipment can read AFS[3] files, then the attacker might be able to make the enabling document and class file visible to the victim by exploiting AFS's global namespace property. *None of the techniques discussed in this paper can prevent applets already available through the client's filesystem from running.*
2. Any scheme that requires examining a TCP stream must understand that stream's encoding. If any browser behind the firewall is able to decompress or decrypt enabling documents or class files, then the firewall must unpack the file in the same way in order to test for offending properties. Naturally, encrypted transfers cannot be parsed by a proxy. Instead, the user's trust in the firewall must be transferred to trust in the remote server. Servers that have public keys delivered with a Web browser can probably be assumed to have higher security standards and accountability than an anonymous cleartext HTTP server; still, a system administrator who feels that the risks of possible applet delivery outweigh the benefits of encryption can always disable encrypted transfers at the proxy.

6. Implementation

We have extended Claunch's `<applet>`-blocker [7] to block CA FE BA BE as well. This proxy was able to block every method we know of passing an applet through the firewall for immediate execution in Netscape versions that were then available. Our implementation does not look for Java class files hidden in Zip archives, but it could be extended to do so.

The base program for both blocking strategies, http-gw (HTTP Gateway), was made available by Trusted Information Systems in 1994 [12]. It was originally written so that

[3] Andrew File System, a product of Transarc Corp.

existing client applications could use it without requiring modification, although the user would have to use the applications differently. In a representative firewalled network, an HTTP 0.9 request for the URL `http://www.ncsa.uiuc.edu/` would be unceremoniously blocked at the firewall. However, the user could instead specify the URL `http://firewall.xxx.com/www.ncsa.uiuc.edu/`. The http-gw proxy running on firewall.xxx.com would receive the request, use the next field in the URL as the real host to contact, and then shuttle the data between the two applications. If the remote host at `uiuc.edu` returned an HTML file, then the firewall would also take care to rewrite its links before presenting it to the client. For instance, the returned page might include a link to `http://www.cern.ch/`; this would have to be written as `http://firewall.xxx.com/www.cern.ch/` to be useful to the client application. HTTP version 1.0 [4] defined a way for client programs to knowingly communicate with a proxy, making the above user adjustment and link rewriting unnecessary; most modern browsers now support HTTP 1.0.

Http-gw proxies HTTP 0.9, HTTP 1.0, FTP, and gopher, all with URL rewriting as required by relevant protocols. It determines the transfer type by examining `Content-type:` headers and filename extensions as described in Section 5.2. Combined with the `<applet>`-blocker and the `CA FE BA BE`-blocker, it is a surprisingly complex and subtle piece of code.

One interesting design decision involved what to do when the `CA FE BA BE`-blocker noticed an applet attempting to pass by. By simply aborting the transfer, the client's Netscape would report a `ClassFormatError`, leaving the user bewildered. Instead, our proxy delivers a *canned* applet that opens a window (subject to the existing screen constraints) with a message explaining that an applet was blocked at the firewall. A little trickery is required here: Netscape is expecting to receive, say, class `PokeHole` and the proxy instead wants to deliver class `CannedApplet`. The proxy proceeds by massaging the `CannedApplet` bytecodes to change its name into `PokeHole` on-the-fly; meanwhile, it must also modify or suppress any *preceeding* `Content-length:` header lines, since the modified `CannedApplet.class` will almost certainly have a different size than the original `PokeHole.class`.

7. Conclusion

Remote execution of code has come into its own with the proliferation of downloadable executable content, Java being the most popular example. However, it is also evident that environments in which Java enabled browsers can download applets from untrustworthy sites are fraught with security concerns. Security breaches in computers that execute content from unknown sites are already great cause for concern. This is particularly so in networks that seek to protect themselves from the rest of the Internet by firewalls because naive implementations operate on the assumption that other local hosts are untainted, and hence more trustworthy. The security implications of a firewall break-in are correspondingly more severe. In such a situation, it becomes imperative to study the security implications of allowing machines to download applets from sites outside the firewall. In our opinion, not enough attention has been paid to this aspect, and this work should serve as a wake-up call to those among us who are not already anxious.

We show in this paper how some common firewalls can be fooled into creating holes using a rather simple Java applet. Once this hole is created, the local host which downloads and executes this applet becomes, effectively, bereft of the security that the firewall affords and, as pointed out above, may be more vulnerable than hosts operating without the assumption of a firewall. Of course, Java is not necessary for this attack, but the fact that Java-enabled browsers and their users are ubiquitous makes it much easier to mount. Ironically, while other security concerns require that the firewall be as clever as possible, this attack makes some of the most sophisticated firewalls susceptible. There is no doubt that blocking applets from crossing a firewall is of essence.

Basically, in order to block applets from crossing a firewall but allow other kinds of content to go through, one has to look for `<applet>` tags in the downloaded stream and delete or replace the applet code. But doing this without killing legitimate traffic is not easy. Almost all the different delivery mechanisms (HTML, FTP, gopher, mail, news) may be used to deliver applets by encapsulating them suitably. We found that looking for `<applet>`, the `0xCAFEBABE` byte signature, and certain filename suffixes are all effective strategies for detecting most of the applet code passing through. We also point out that, because of the fragmented nature of IP packets, detecting signatures can be done with more facility in a proxying environment than with a pure packet filtering mechanism. Even in a pure packet filtering environment, some application-level proxying should be considered. Using the strategies we describe in such an environment, one can be more confident that a Java applet cannot pass through the firewall. The implementation of these strategies, starting from Claunch's `<applet>`-blocker, is effective if somewhat subtle.

Attacking firewalls using Java applets is only the tip of the iceberg and the authors believe that more vulnerabilities will be found in the future involving firewall security and Java. What we have proposed can only be the first step towards creating a secure environment for executing downloaded content using a firewall without sacrificing the new dimensions in computing that Java avails. Problems involving securing networks from malicious code from the Internet abound. In particular, how to protect against portable executable formats like Javascript that somehow beat the `<applet>`-style blocking mechanism remains an open problem.

Acknowledgments

We would like to thank Carl Claunch and Ed Felten for useful discussions, Steve Bellovin for providing valuable feedback and corrections, and the reviewers for their comments.

References

[1] F. Avolio and M. Ranum. A network perimeter with secure external access. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1994.

[2] S. Bellovin. Personal communication.

[3] S. Bellovin. Firewall-friendly FTP, 1994. RFC 1579.

[4] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol, 1996. RFC 1945.

[5] D. B. Chapman and E. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates Inc., 1995.

[6] W. Cheswick and S. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.

[7] C. Claunch. Java, JavaScript and ActiveX Screening Modification, 1996. Available at `http://www.hdshq.com/fixes/fwtk/`.

[8] D. Dean, E. Felten, and D. Wallach. Java Security: From HotJava to Netscape and Beyond. In *IEEE Symposium on Security and Privacy*, 1996.

[9] E. Felten. Personal communication.

[10] R. Ganesan. Bafirewall: A modern design. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1994.

[11] D. Hopwood. Security bugs in Java. Available at `http://ferret.lmh.ox.ac.uk/~david/java/`.

[12] Http-gw, a gateway for HTTP. Part of the fwtk firewall toolkit by Trusted Information Systems Inc. Available at `http://www.tis.com`.

[13] The Java Virtual Machine Specification Version 1.0 (beta). Available at `http://java.sun.com/doc/language_vm_specification.html`.

[14] E. Messmer. Corporations to pave streets with Java gold? *Network World*, June 10, 1996.

[15] J. Mogul. Simple and flexible datagram access controls for Unix-based gateways. In *USENIX Conference Proceedings*, 1989.

[16] R. Ranum. A network firewall. In *Proc. World Conference on System Administration and Security*, 1992.

[17] J. Reynolds and J. Postel. File transfer protocol (ftp), 1985. RFC 959.

[18] J. Reynolds and J. Postel. Assigned numbers, 1994. RFC 1700.

[19] C. Stoll. *The Cuckoo's Egg*. Doubleday, 1989.

[20] W. Treese and A. Wolman. X through the firewall, and other application relays. In *USENIX Conference Proceedings*, 1993.

Internet RFCs are published by the USC Information Sciences Institute at Malrina Del Rey, CA, and can be obtained from `http://www.isi.edu/publications.html`.